Novel User Interfaces for Diagram Versioning and Differencing

Darius Dadgari, Wolfgang Stuerzlinger
Dept. of Computer Science and Engineering, York University
4700 Keele Street, Toronto, Ontario, M3J 1P3, Canada
http://www.cse.yorku.ca/~wolfgang

ABSTRACT
Easily available software for diagram creation does not support the comparison of different versions and the merging of such versions. We present new methods and techniques for easy versioning of general two-dimensional diagrams. Multiple novel versioning methods for diagram versioning are compared to each other as well as to previous work in a user study. Participants in a user study preferred the Translucency View and Master Diagram Scenario to the other investigated methods and scenarios.

Categories and Subject Descriptors
H.5.2 [**User Interfaces**]: Graphical user interfaces.

General Terms
Human Factors, Design, Documentation.

Keywords
Version control, merging, dynamic views, translucency.

1. Introduction
From their inception, vector-drawing programs have consistently increased in their functionality. Partly based on growing memory, storage, rendering and facilities of newer and better computers, the available operations have been extended and the user interface improved to enable more obvious and intuitive ways to perform complex tasks. The goal is to allow users to do more while requiring less knowledge on their behalf. This is certainly useful with respect to ease of use, but there are few verifications of the efficacy of these improvements.

Diagram creation is a common task, and frequently used in the areas of architecture, modeling, engineering, design, information visualization, concept diagrams, software engineering, and many other areas. Most such diagrams evolve naturally over time. Hence, diagram versioning is a common task that compares two (or more) versions of a single diagram to create a new version, to synthesize the "best" features of the old ones. Versioning is part of common activities, such one or more persons modifying a Computer Aided Design (CAD) drawing to better suit an architectural requirement. Another example is a teacher creating a diagram and modifying it later to suit a different educational goal.

British HCI '10, Month 1–2, 2010, City, State, Country.

Functionality for versioning has slowly been incorporated into multiple application domains. Text document versioning systems are widely used and many include support for versioning. Source code versioning is often used when collaboratively coding computer programs. Even though these systems are currently available, there is no published comparative examination of these methods and the usefulness of these versioning methods is not widely known. Versioning tasks for diagrams are currently often performed without the aid of a computer, especially since there is almost no software support for versioning of diagrams in easily available drawing systems.

This paper presents several new techniques for Diagram versioning. More precisely, we investigate techniques that allow users to perform the task of selecting desired parts from different versions of a diagram. We limit ourselves to graph-like diagrams with nodes and arcs that connect nodes. The results may be generalizable to other visual creation tools or diagram types; however, their applicability has not been directly examined. We target ease-of-use as well as user interface efficiency. Unintuitive techniques provide little benefit for the average user regardless of the efficiency potential. Moreover, we validate these techniques with user studies.

1.1 Related Work
Diagram versioning, especially for structured diagrams, has been studied previously, but mainly in software-architectural diagramming [15]. A recent survey can be found in [3]. Ohst *et al.* [13] examined versioning methods for UML class and object diagrams. They also proposed guidelines for diagram versioning methods, such as showing the common components only once, and providing a method that uses color differentiation to differentiate between versions. However, this work focuses on class and object diagrams, more specifically UML diagrams, and their methods exploit the rigid structure of those diagrams and the fact that spatial layout is (mostly) irrelevant in this domain. Ohst's methods do not translate well to most other two dimensional diagrams, as in general diagrams the location of nodes and links can represent important information to the viewer. Although research into different layout schemes for UML diagrams exists [1], this research only focuses on how to best present different types of UML diagrams.

The diagramming application in the Marama collaboration tools [4] allows for collaborative, asynchronous work on general diagrams. These tools provide an implementation of diagram versioning. However, the intent is not to reconcile multiple versions of a diagram, but rather on how to let other users know what is potentially changing or what particular information another user is looking at in asynchronous collaboration. Differences between diagram versions are shown in text form only and users are encouraged to collaborate with each other to reconcile the differences. One method designed to encourage

users to reconcile manually the ideas and potential differences is to use 'sketching' on a temporary overlay on the content.

Using text to describe differences between diagrams in textual form is a very primitive method. For example, minor changes in position or size would be presented as significant with common tools such as the Unix `diff` command. A step forward is the Pounamu system [11],[17], which introduces a generic method to perform general diagram differencing. Users are presented with a text list of the differences and can go through the list of changes to determine which changes to accept and which to reject. A simultaneously displayed graphical representation of the diagram visualizes the changes. Varying background highlighting depicts the amount of changes in a node. Solid and dashed lines are used to represent additions or deletions relative to the previous version. This work also included a "formal user survey", however the evaluation only examined this one newly proposed method (i.e. did not compare it to any other system), and only examined UML diagrams. Further, no statistical analysis of the survey results was provided. The survey on Pounamu contains an interesting statement regarding the presentation of difference information: "Our approach has some limitations. As indicated above from our user survey … users cannot control how changes detected by the differentiation plug-in are presented … this is somewhat frustrating to users." Furthermore, a topic for future work included "providing users with the ability to change this highlighting used by the highlighting plug-in", suggesting that the current highlighting method could be improved and that other methods may prove to be more beneficial for diagram versioning.

Ohst *et al.* [12] discussed the problems of overusing highlighting in diagrams. This approach attempts to minimize the number of highlighted elements when examining differences – as parallel changes on separate versions quickly leads to a large number of differences. Highlighting may then simply color the entire diagram space, and thus be (almost) useless. To counteract this, the authors propose that users can select which differences in the document can be colored – either logical changes or structural changes. This method may potentially function well for expert users, which understand the differences between these two kinds of changes, but will likely be problematic for users without a technical background.

Work on undo and operation histories is related to versioning, see e.g. [8],[9]. However, such systems support only a single document, and do not permit differencing and merging of versions. A side-by-side reconciliation method, e.g., as in WinMerge [16], may seem at first to be a reasonable way to present different versions of a diagram. However, this idea works well only because of the linear nature of text, which permits the (arbitrary) insertion of spacing to synchronize versions. As spatial positions matter in most diagrams, synchronized side-by-side views are much less realistic for diagrams and hence limit the applicability to this domain. Finally, we point to a recent survey on software merging techniques [10] for techniques related to merging different versions of code. However, most of these methods are code/text-based and hence do not apply directly to diagram versioning.

2. Diagram Differentiation Techniques

In the following, we present and examine multiple view control schemes for diagram versioning. These methods allow users to compare multiple on-screen versions of diagrams and select which portions of the presented versions they wish to include into the result. Some methods are similar to text versioning methods, but have been tailored for the diagram domain. We present and discuss potential view techniques to help facilitate diagram versioning and discuss interaction methods.

View control methods are designed to help facilitate viewing the relations between versions. These views are accessible to users via dropdown menus within an editor. Views are separated into two menus: viewing accepted items and/or rejected items, and viewing relations between the diagrams. Views from both these menus can be mixed-and-matched. For example, a user who wishes to view only rejected items that are similar between all versions of the diagram can select the corresponding options. Likewise, the user can also choose to view only the items, which still require their attention (items which have not been accepted or rejected). All these options are explained below. Figure 1 shows two versions of a diagram. Figure 2 shows both versions simultaneously, with the two versions color-coded in red and blue respectively. Common parts are shown in black. This figure also depicts the visual appearance after several accept and reject decisions have been made.

We are aware that using color to visually differentiate versions can be problematic if the diagrams use color to convey meaning. However, we do not know of a better approach as the diagram itself may use any particular combination of color or other visual features. One option to avoid this is to change the layout of the diagram. However, if the layout contains semantic information, any change to the layout may change the meaning of the diagram. Consequently, we cannot offer a perfect solution, but allow the user to configure the version colors to cover most situations.

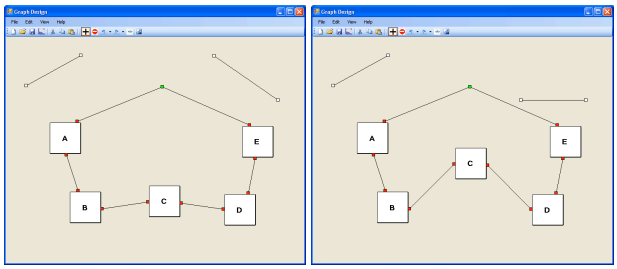

Fig. 1. Two versions of the same diagram (Left: X, Right: Y).

In addition to the specific diagram versioning methods described below, users are able to accept or reject changes on the graphs provided. Users are also able to choose whether or not to view those accepted and / or rejected items. Upon initialization of the diagram comparison, an automatic algorithm processes the graph. Graph objects in both versions that have the same size, position and the same attributes are automatically considered accepted. No diagram object is considered rejected initially. For a user to accept or reject graph objects, they need only to select them by right-clicking on an object. Users also have the option of clicking and dragging over an area then right-clicking on the screen to choose whether to accept or reject all items in that area via a right-click context menu.

Accepted items are shown by default in most views, and the ability to view rejected items can be enabled via a drop-down menu. Both accepted and rejected items are color differentiated from all other views, and supersede all other views. Thus, if a user has chosen to show accepted and/or rejected objects, they will

always be shown foremost over any other layer. This is to ensure that users do not forget or are unable to see objects on the diagram that they have already accepted.

With the Accepted and Rejected items menu users can enable or disable the Accepted or Rejected views. This allows users to toggle the display of accepted items, and the visualization of rejected items. The ability to disable both Accepted and Rejected items permits the user to view which graph items – if any – still require their attention without presenting any information extraneous to that task. The usefulness of this view depends on how much information has already been accepted or rejected. For self-contained diagram objects or when two choices are clearly denoted, this method may focus attention to certain objects. Due to the potential usefulness of this method, we provide one-click access to this as "Differences between Layers". Viewing rejected items can help to focus attention on what objects were either wrong or have been improved. Viewing these older, errant objects can potentially help further determine what can be improved. The rationale behind viewing accepted items is self-evident.

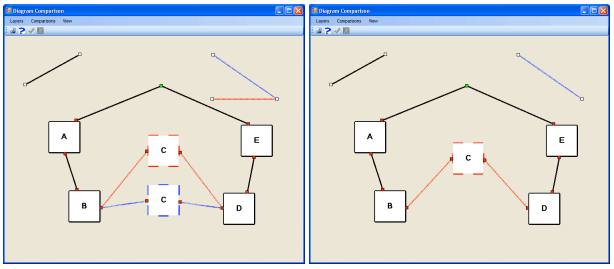

Fig. 2. (Left) X and Y in versioning mode in the default view. (Right) View after rejecting node C and the connecting links from X (blue), as well as the upper-right link from Y (red).

Note that the various options to show accepted or rejected graph items do not change the basic view of the relations between diagrams. They only add or subtract information that would be provided in addition to those algorithms. As such, users can always be guaranteed a level of output consistency when using the relation views – regardless of what other options are selected. In addition to the Accepted and Rejected items menu, users need ways to view the relations (similarities, differences etc.) between versions. Without these options, determining which portions of a diagram need to be accepted, modified, or deleted becomes much more difficult. We provide multiple different views of the diagram comparison to help users select which portions of the diagram to accept or reject.

2.1 Similarities between Layers

The ability to view only the items that are simultaneously in both diagrams is important. This effectively allows the users to determine the 'core' of the diagram between versions, which provides the context for determining which portions of the diagram that are not similar, should be accepted or rejected.

Similar to other current diagram versioning implementations, we determine similarities and differences between versions. The original `diff` tool compares the input of two versions of text data and produces output of all changes on that data. When comparing drawings that were created independently and/or with objects stored in (potentially) different graphs this is not sufficient. However, as we examine only diagrams in this work that are incrementally modified, we use the history of objects to match them in a simple, yet reasonably reliable manner, i.e. we perform an operation-based merge. For the generalization to arbitrary diagrams, one would have to match objects based on some measure of similarity, including visual similarity. However, visual similarity is a problem that is not well understood in general, see e.g. the state of the art in object recognition, and we do not attempt to address this complex problem here.

A straightforward matching process does not incorporate any tolerance for values that are similar but not identical, which causes problems with diagrams, where locations that are not exactly pixel-similar between versions are considered different by a basic differencing algorithm, yet for general diagrams these near-identical graphical items may likely be functionally the same. Consider also the case of UML diagrams, where positions are irrelevant so long as data links are the same. Hence, our similarity function allows the user to select the following options upon creation of a diagram:

- Only pixel-perfect locations are considered similar
- An isomorphic method, where only the structure of the graph matters, not positions.
- A level of positional tolerance where otherwise identical graph items in the same general location are considered similar.

The rationale for the pixel-perfect method is that the position of diagram nodes and links may provide different information to the user regardless of the semantic similarity. For example, a pyramid shaped diagram and a diamond-shaped diagram may be isomorphic. The diagram designer however may have created that shape to convey some level of meaning. Due to this, we ensure that semantic similarity is an option that the user can choose to select according to their needs for that particular diagram. The relaxed method is also necessary for incorporating diagrams in which positions are irrelevant, and the node-link structures contain the graph information (UML etc.). As diagrams can be changed either accidentally or purposefully to a degree that is virtually imperceptible to humans, it is important to provide a less strict matching criterion. Take for example a diagram on a 1600x1200 pixel space where one node has been shifted one pixel to the left. This is practically visually imperceptible and the node movement does not really convey visual information to the viewer. Therefore, both diagrams can be considered contextually the same from a users' perspective. However, determining when a change on a diagram shifts from virtually imperceptible to perceptible is a difficult task. As a simplistic example, consider that a change by one pixel may cause a moved node to overlap a link. This may change the perceived meaning of the diagram completely. Although this problem is interesting, it is outside the scope of this document. Fig. 3 top left shows the Similarity view.

2.2 Differences between Layers

The opposite of the Similarity between Layers view is the Differences between Layers view, as shown in the middle part of figure 3. This view allows users to explicitly see those graph items which are dissimilar between all versions of the graph. Moreover, all accepted and rejected graph items are not shown by default in this view. Thus, this view presents to the user only those objects that still require their attention, which provides a very quick way for users to see only the parts of the diagram, which need to be reconciled, or where the changes lie between two versions before

merging. When examining this view the user also immediately see all changes. This is particularly useful when versioning between different people as the context of changes can be seen more easily.

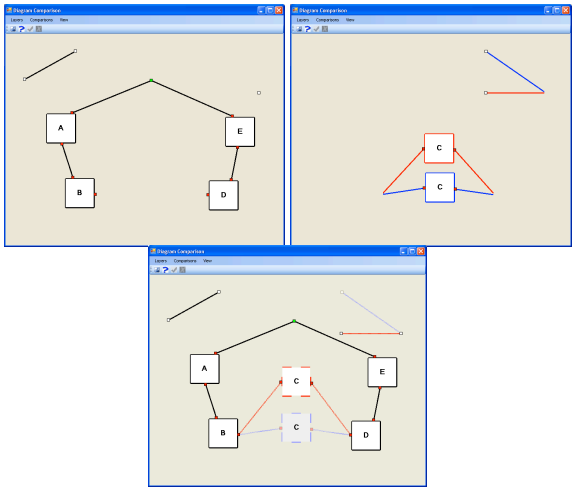

Fig. 3. (Top Left) Diagrams X and Y in the Similarities between Layers view, where only items in both diagrams are shown. (Top Right) The Differences between Layers view, which displays items different in both diagrams. (Bottom) Translucency view on diagram Y.

2.3 Translucency View

A translucent overlay view, similar to the basic layer view control, allows the user to view multiple versions as individual layers at a time. This overlay view uses alpha blending [14] to enable users to view multiple versions while decreasing the focus on the "main" version. The Overlay method provides a middle ground between completely showing a version and completely hiding it. However, objects or links cannot be moved in this mode, as this breaks the consistency between the shown layers. Hence, we implemented a new method, where *only* the 'top' layer is translucent, whereas the 'bottom' layer is fully visible, i.e. fully opaque. This enables display of objects in both layers, even if they are at the same position. Users may click on and use both the fully visible as well as translucent objects to perform versioning tasks. This is implemented via click-through on the top layer: if a user selects a location where there is no object on the top layer, and there is an object on the bottom layer, the object in the bottom layer object is selected. The bottom part of figure 3 shows the Translucency view.

2.4 Enabling and Disabling Layers

We provide users with a menu to select which layers are visible or invisible. This is necessary, as changes in different versions can occlude each other. Even in the overlay view this leads to significant on-screen clutter. Hence, it is essential to be able to view only particular layers at a single time. For completeness we point out that disabling a layer still retains all accept/reject decisions on the respective graph items.

3. Diagram Comparison Scenarios

To help users spend less time in versioning, we explore the idea of automating certain tasks. One central concept here is a 'Senior' or 'Master' diagram that is versioned with other diagrams. We compare this to a situation where all versions are given equal weight, which we call the 'Parallel' scenario. Depending on the situation, either one or the other is more appropriate.

Designating a diagram as master has benefits. For example, in a situation where an agreed upon previous version exists, any reject decision on a particular modification means that the corresponding object in the master diagram is automatically accepted. Effectively, this gives precedence to the master diagram – at least until a new version is designated as the new master. We have termed this the Master Diagram Scenario. Often, the nearest common ancestor is used as master in document merging when multiple people are involved. We implement this functionality by permitting users to designate a given version as master via a drop-down box.

Conversely, in a situation where a single user is creating multiple alternative diagrams, none of which is inherently preferred to another, all versions are equally viable. Another example is a group of users are discussing potential ideas amongst themselves. We call this the Parallel Diagram Scenario. This mode is active whenever no version is designated as master.

Last, but not least, we point out that it is possible that multiple people can change a diagram independently in parallel. While this is an interesting topic, an overlay of all changes produces a very cluttered view. In the text domain this can be addressed with different text colors and by showing multiple changes in sequence. In the diagram domain, and without the ability to change the layout of the diagram arbitrarily, changes quickly overlap and important details become obscured. Hence, we believe that pair-wise diagram comparison is the most reasonable approach, even if there are multiple, parallel changes.

4. Diagram Editing Modes

Another facet of versioning concentrates on the editing operations that can be performed on the various versions. Some implementations permit users to edit multiple versions at the same time and allow users to accept and reject changes simultaneously. The most prominent example for this is most modern text editing software. Other implementations of versioning require users to accept and reject all changes before normal editing can resume. While this distinction looks superficially similar to user interfaces with and without modes, the issue goes deeper as a different style of work and decision-making process is associated with each choice. The option of "checking in" changes is not appropriate in situations where decisions are made in directly in a meeting. To support all situations, we implement a toggle between both alternatives and discuss them below as the Decision Editor and Flexible Editor mode.

4.1 Decision Editor Mode

In the decision editor mode, users reconcile multiple versions of a diagram prior to any further modifications on that diagram. Preventing users from editing the diagram before determining which version is desired is useful when there is a need to select which changes must be accepted prior to performing another iteration of the diagram. This task arises with blueprint schematic creation, and other architecture and professional design systems. Further, in many engineering design disciplines, a core principle is to reconcile current work before continuing with new work. Preventing modification during version merging also ensures that

decisions are made before further changes are done, which helps with documentation and other organizational issues. This mode is also representative of many Computer-Supported-Collaborative-Work tasks, in which multiple users are working together on a single system to reconcile changes at once.

In our implementation, whenever the user decides to compare two versions of a document, a new versioning Decision window appears in the same position as the editor window and the editor window is closed. The Decision window permits only access to all operations related to viewing the versions, layers and all other toggles described above. Moreover, an 'Accept Graph' and 'Reject Graph' buttons become available, as well as 'Final View'. Accepting the Graph returns the user to the Editor window. Rejecting the Graph returns the user to their previous view to perform more decisions in the Decision window. Users can then use the layer views and options to reconcile versions as needed. However, since the editing tools are unavailable, users cannot actually modify the diagram. Users can only select whether the diagram components are to be included in the new version.

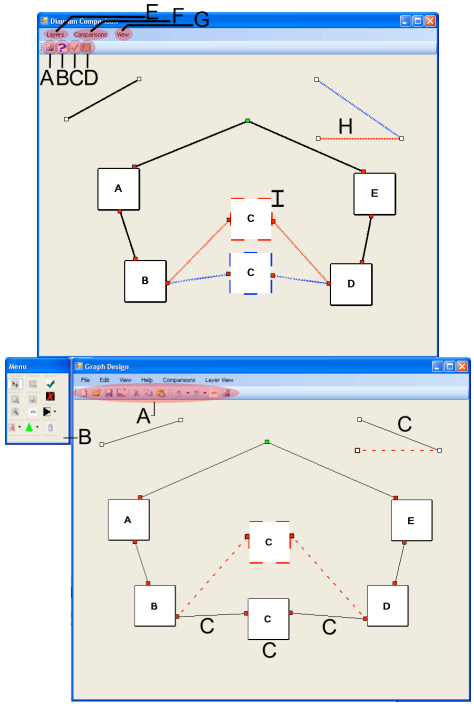

Fig. 4. (Top) Decision Editor. Big letters denote options available, see text. (Bottom) Flexible Editor.

Once the user is satisfied, they can click 'Final View', which automatically changes the diagram view options to only show all accepted parts. This provides users with a view of the exact diagram that they have approved, which they can accept or reject or toggle back to continue with the diagram rectification. This overall functionality is representative of the 'Yes/No/Cancel' options in most software packages.

In the top part of figure 4, the letters illustrate the various user interface elements in the Decision Editor mode. 'A' is the transparency toggle, which toggles transparency of the top layer, in this case blue. 'B' is the Final View button. 'C' and 'D' are the Accept Graph and Reject Graph buttons respectively. 'E' toggles the visibility of the graph layers that are being compared. 'F' allows the user to toggle the ability to see only the similarities respectively the differences of the graph. 'G' toggles whether to view only accepted items, only rejected items, both, or neither, where the last option shows only the graph items that still need a decision from the user. 'H' denotes that if the red line is right-clicked, a context menu pops-up which allows the user to decide whether to accept or reject the line. If so, then the line is removed; however the right link connecting the line remains, since the right-most link is connected to the blue line. 'I' denotes that if the red node is right-clicked, a context menu also pops-up which allows the user to accept or reject the node. In this case, if the node is rejected, then the node, the two red links connected to the node, and the lines connected to the node are removed. Most of this functionality is common in modern graphical user interfaces.

4.2 Flexible Editor Mode

The Flexible Editor allows users to reconcile multiple versions of a diagram while still also being able to use all editor functions on the diagram workspace. This gives the user more flexibility and encourages exploration. Consider as an example a scenario, where the user has a new idea during the reconciliation of two versions of the diagram. In this situation, access to editing features may be crucial. In summary, the Flexible Editor mode allows users to reconcile and edit concurrently, without having to decide at any particular stage which version will be considered accepted.

In our specific implementation, the user does not have to switch modes in order to use versioning or editing functionality. Instead, all operations are available at all times. If the user asks for a comparison of versions, the comparison algorithm is run on the current version and updated results are shown on the screen. Initially, any new objects cannot be moved or used. However graph items can be accepted or rejected by right-clicking on any diagram object and selecting accept or reject from the context menu. Upon acceptance, that object becomes part of the document shown in the editor working space can now be modified with the editing functions. A single acceptance of an object will ensure that the object cannot be rejected – unless an Undo action is performed subsequently. The object can still be deleted as normal.

In the bottom part of figure 4 the letters illustrate the functionality in the Flexible Editor mode. 'A' points out that all view functionality (including 'Save') is available even while there are objects that have been neither accepted nor rejected, i.e. the next iteration of versioning is not complete. 'B' illustrates that normal editing functions are available to the user. 'C' shows an example of the Master Diagram Scenario for diagram versioning, where Diagram Y is automatically accepted upon comparison.

5. Basic versioning editor features

All of the above mentioned functionality was implemented in the DIAGEM system. The system was created in Visual Studio .NET using the Visual Basic 6 runtime.

In order to examine the presented diagram versioning methods, we implemented basic diagram editor functionality, such as creating, selecting, moving, and deleting objects, as well as undoing actions. Moreover, the system provides users with the ability to view multiple distinct versions of a diagram all at once in a single editor program. The software allows users to create nodes and link objects, with squares or arrowheads visualizing the direction of the links. The color of the squares or arrowheads

encodes attachment states. Red on both sides shows that the link is attached to nodes on both sides, green on one side denotes the link is not attached on that side, and white on both sides to denote that the link is not connected to any nodes. Attached links move to the same relative position on a node as the node is moved or resized. Link snap to any nearby node upon creation and then also automatically attach the link to that node. Other basic tools in the editor include one-click creation of nodes, a rubber-banding tool for creation of links.

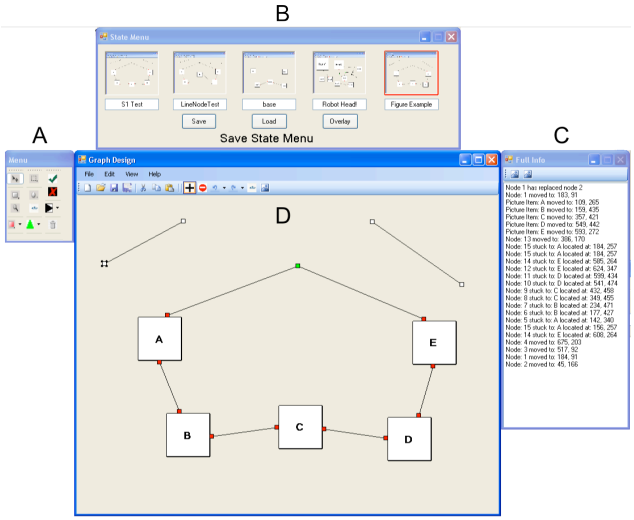

Fig. 5. View of basic diagram versioning. Letters denote user interface modules in DIAGEM, as explained in the text.

In DIAGEM we allow users to perform operations on up to six versions of a diagram at a time. Moreover, we provide a history panel with five spaces for different (named) versions, for easy access to versions as well as easy saving and loading of versions. To help users remember the content of versions, the system provides also a small picture of each version. While this is superficially similar to undo histories [8],[9] and features in Adobe Photoshop, we point out that in contrast to undo, users *explicitly create versions* in versioning, usually when changes that have some significance to the user have occurred. Users usually also name versions to underline their significance. Explicit creation of versions also helps with the problem that entries in undo logs do not correspond necessarily to meaningful states in the terms of the user, even if events are compacted automatically. User interfaces for undo, including the 'History Brush' in Adobe Photoshop, are also usually more lightweight than versioning. But a fundamental difference is that undo does not persist across sessions. Finally, we expect most applications of diagram to use only two or three versions at a given time, as visual clutter increases significantly when many graphs are shown, which hinders a users' ability to perform tasks [11].

DIAGEM is pictured in figure 5. Window 'A' is the diagram creation and modification palette that enables users to create and operate on nodes and links. 'B' is the history panel that allows users to name and preview their saved diagrams. Window 'C' provides a text log of all performed actions. This is mainly used for debugging, but can also be used by users to enable recall of what was previously done. 'D' denotes the main working area where objects are created, reconciled, deleted and otherwise modified. Context sensitive right-click menus are available in this window. For example, right-clicking on empty space gives users to access to common editing operations such as cut, copy, paste, undo and redo. Right-clicking on an object provides access to modification operations for that object, such as changing the color or size, in addition to the other editing operations.

Colors are also used to differentiate between versions. A colored dashed line around the boundaries of nodes is used to visualize which version it belongs to, see e.g. figure 2. Similarly, colored dashed lines are used to visualize links that are in a given version. Dashed lines were chosen so that users can more easily differentiate between versions, compared to the solid lines that visualize links identical between versions. As shown in figure 6, the choice of dashed lines was also motivated by the fact that two dashed lines can visually "meld" to create a more or less solid line, which helps to visualize that two links are very similar between versions.

Fig. 6. Example of dashed-lines for links between nodes. Each color represents a separate layer. Note that the overlay of the red and blue dashed lines can create a solid line.

6. User Study

The goal of this study was to examine the efficacy of the diagram versioning techniques described above. If the addition of diagram versioning tools provides benefit to users that perform these tasks, this would suggest that these tools should be added to commercial systems, and that further research into expanding the usefulness of these versioning tasks is worthwhile.

In order to identify which methods users prefer, we provided the following relatively informal evaluation scenario: Due to increased student population, the local university has decided to build a small-scale shopping mall on campus with several kinds of stores, such as food, entertainment, shopping, postal service. Participants are then asked to build a diagram of the shopping mall and choose which stores to include in the mall. Beyond the stores, each store has up to two 'specialties', which are partner deals between stores. An example is that if you purchase an item at store 1, you will receive a coupon for store 2.

Stores are denoted as node and the size of each node is proportional to the size of the store. The proximity between stores and general shape of the shopping mall is visualized as the distance between nodes and the relative position of nodes, and the partner deals are denoted as links between nodes, see figure 7.

To simulate a progression of work, participants followed a script, which contained a total of 35 relatively short tasks. The start condition is depicted in figure 7. In each task participants were asked to modify their diagram based on some changes in the university's needs. An example is: "It has been six months since you have opened your mall. Unfortunately, a new competing shopping centre has been built nearby and has taken some of your business. To maintain your profit you will need to close two stores and remove two of your specials". Each participant was given the same questions in the same order to ensure consistency. However, participants were free to choose which stores and specials to remove, which provided a semblance of free choice. Participants were asked to keep track of previous iterations of their mall using the save facilities in the history panel. After a sufficient number of

relatively simple tasks, participants were then asked to revert portions of their mall to its previous state using the versioning functionality. An example for this is the following instruction: "Due to a huge jump in the number of students enrolled in graduate school, a year later your mall is doing much better than before and you are ready to re-open some of the stores you previously closed to save costs. Please re-open two of your stores and add a special".

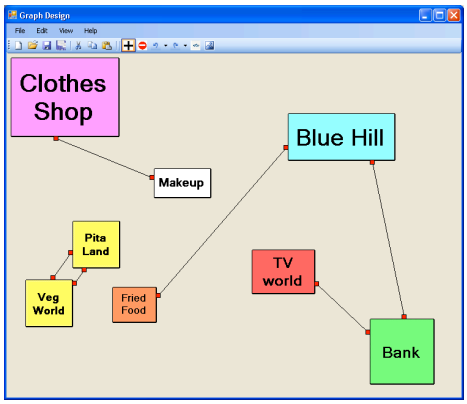

Fig. 7. Example of a default mall design. This design is iterated upon by the user and slowly modified using the versioning tools to become completely different by the end of the study.

During the study, users were required to use all mentioned methods and views for diagram versioning. The experimenter kept track of which methods were used for each task and balanced their usage by forcing users to switch to another method or view for the next task, if they had used one method or view more than proportional. Essentially, this ensured that all participants used all conditions with almost equal frequency. The motivation for this is that we wanted to give participants some freedom of choice as to the methods and views, while still ensuring that enough data was collected overall. A counterbalanced design was considered, but was judged impractical, as it would require more participants.

Eight participants, three female, aged 25-33 with mean 27.1, were given a default diagram layout of the mall, with multiple stores and specialties already created. The experiment completion times were between 40 and 65 minutes. This time includes five minutes of practice prior to the study. Upon completion of the experiment, users were given a five-point Likert scale questionnaire containing questions such as "How useful was the (name of method) method for versioning?" and "Which methods did you prefer (circle all that apply)". Afterwards, an informal discussion between the participant and the experimenter was used to solicit feedback on the various diagram versioning methods.

Our general expectation for this experiment was that performing diagram versioning with a user interface similar to other versioning systems, such as Microsoft Word, with a user interface targeted at versioning would enable users to quickly use the system. The methods and scenarios described in the beginning of the paper reduce clutter, highlight important items, permit the user to instantly view previous versions of the diagram and for the case of the decision editor, and provide a method to ensure users can explicitly know at what steps they are in the versioning process. As the experimental design involved participants following a script with some freedom in the decisions, we did not record timing information and/or errors. Moreover, the decision making process on some tasks would confound timings in any case. Hence we chose to focus only on completion of the scenario, observations and ratings on the questionnaire.

The versioning techniques in the DIAGEM system as described above were evaluated on a computer with two 19" displays at 1024x768 resolution each. A keyboard and mouse were used as input devices to the system.

During the experiment, we created a protocol to record which versioning method was used for each task by each individual participant used. We also recorded whether the results of a given operation were useful to the participant. Given that all participants followed the same rules outlining the numbers of stores and specials, all performed essentially the same experiment. While the users chose positions, names, and specials freely, the structure of the produced graphs was similar through the set of instructions they followed. This enabled us to examine each stage with a high level of consistency between users.

6.1 Results

All participants were able to complete the scenario without problems after the initial training. With a few minor exceptions in some instances, participants did not require significant help during the tasks. We see this as a confirmation that the basic ideas for versioning of diagrams presented here work well enough for naïve users. In a few, rare, instances, participants did not follow the script closely enough and the experimenter had to intervene.

The questionnaire results provided us with a clear sense of user preference. Participants strongly preferred the Flexible editor mode compared to the Decision editor, with a median rating of 4 for the Flexible editor and 2 for the Decision editor. Comments from participants stated that they did not like to be 'tied down' to making decisions without the ability to edit, as they did not always wish to just focus on versioning tasks and then afterwards continue normal editing. As previously mentioned, the Decision editor mode is likely better suited for tasks in which a decision *must* be made prior to any editing. It should not be used otherwise.

Users particularly liked the Translucency View with a median rating of 4. However, participants used this functionality in a different way than expected. Many users just enabled the transparency effect with an older version of the diagram and left it on-screen even while performing completely new additions to the diagram. When asked if this was done inadvertently, one participant responded: "I wanted to see what I did before, so I just left [the translucency effect] on". Translucency provides users the ability to keep previous iterations of the diagram in mind, without forcefully focusing users' attention onto that previous iteration. As such it was strongly preferred.

We also discovered that users strongly preferred the Master Diagram Scenario, with a median rating of 4. The Parallel Diagram Scenario had a median of 2 and a mode of 1. Most comments related to the usefulness of using this functionality as a shortcut to specify which aspects of a diagram to accept upon an object rejection. Another way to state this is that users preferred to use methods that involved fewer unnecessary steps.

The Similarity and Difference views were not used as often as the Translucency view. The experimenter often needed to ask participants to use these views while diagram versioning, since for

most participants these views took a back seat to the Translucency View. Users preferred seeing all information on-screen at a time as opposed to hiding information that was not pertinent to them. It may be possible that users will learn to use these methods with more experience. Although they may not be as readily intuitive, they can provide an unobstructed view of the portions of a diagram that require attention, without the need to view portions of the diagram that are already completed.

Interestingly, many participants inherently felt that versioning techniques were similar to the well-known undo functionality. People would casually tell the experimenter that they were going to 'undo' changes, when they were actually comparing multiple versions of a diagram. Undo operations are similar to versioning tasks as rejecting a modification on a version is an undo action upon that modification. But undo is different in the sense that it works at the level of individual primitive actions or (simple) sequences of such, whereas the user creates versions explicitly upon the completion of significant parts of work, i.e. at a much higher level of granularity, essentially based on semantics.

7. Conclusions

Multiple methods and techniques for diagram versioning were described, as well as the results of a user study to examine user preference. Based on reflections about how users can use versioning we presented several methods for visualizing versions and the differences or similarities between them. In particular, the Translucency view offers easy access to a previous version without focusing the user's attention overmuch onto this older version. Moreover, we discussed different modes for editing, which either force users to make decisions before progressing or give the user the flexibility to mix decisions and editing. We also performed a user study to determine which versioning techniques users prefer. We found that users particularly like the Translucency View and Master Diagram Scenario.

In future work, we are planning to follow up on the connection between versioning and undo. Moreover, we are planning to investigate the use of re-layout methods, which naturally depend on the application domain, to address the problem of overlapping, yet different elements.

8. References

[1] Andriyevska, O. Dragan, N. Simoes, B. Maletic, J. Evaluating UML Class Diagram Layout based on Architectural Importance. In Proc. VISSOFT, 2005.

[2] Couture, B. Rymer, J. Discourse interaction between writer and supervisor: A primary collaboration in workplace writing. In Collaborative Writing in Industry: Investigations in Theory and Practice, 1991, pp. 87-108.

[3] Förtsch, S., Westfechtel, B. Differencing and Merging of Software Diagrams: State of the Art and Challenges, ICSOFT Workshop on Comparison and Versioning of Software Models, 2008, pp. 7-12.

[4] Grundy, J. Hosking, J. Huh, J. Li, K. Marama: an Eclipse meta-toolset for generating multi-view environments, Formal demonstration paper, Conference on Software Engineering, May 2008, pp. 819-822.

[5] Harrison, B. Kurtenbach, G. Vincente, K. An Experimental Evaluation of Transparent User Interface Tools and Information Content. In Proc. UIST 1995. pp 81-90.

[6] Horton, W. Overcoming Chromophobia: A Guide to the Confident and Appropriate use of Color. IEEE Transactions on Professional Communication, 34(3), 1991. pp 160-171.

[7] Hunt, J.W. McIlroy, M.D. An Algorithm for Differential File Comparison. Computing Science Technical Report #41, Bell Laboratories, 1975.

[8] Kurlander, D. Feiner, S. A Visual Language for Browsing, Undoing and Redoing Graphical Interface Commands, Visual Languages and Visual Programming, 1990, pp. 257-275.

[9] Meng, C. Yasue, M. Imamiya, A. Mao, X. Visualizing Histories for Selective Undo and Redo, APHCI 98, pp. 459-464.

[10] Mens, T. A Sate-of-the-Art Survey on Software Merging, IEEE Trans. On Software Eng. 28 (5), 449-462, May 2002.

[11] Mehra, A. Grudny, J. Hosking, J. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. ASE, 2005. pp 204-213.

[12] Ohst, D. Welle, M. Kelter, U. Differences between Versions of UML Diagrams. In Proc. Foundations of Software Engineering ACM, 2003 pp. 227-236.

[13] Ohst, D. Welle, M. Kelter, U. Difference Tools for Analysis and Design Documents. In Proc. International Conference on Software Maintenance, 2003.

[14] Porter, T. Duff, T. Composting Digital Images. Computer Graphics 18(3), 1984. pp 253-259.

[15] Westhuizen, C. Hoek, A. Understanding and propagating Architectural changes. 3rd Conference on Software Architecture IEEE/IFIP, 2002. pp. 95-109

[16] WinMerge differencing and merging tool. Further information available at http://winmerge.sourceforge.net

[17] Zhu, N. Grundy, J. Hosking, J. Liu, N. Cao, S. Mehra, A. Pounamu: A meta-tool for exploratory domain-specific visual language tool development. Journal of Systems and Software, Elsevier, vol. 80, no. 8 pp 1390-1407.